

Algoritmos y Estructuras de Datos II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

TP2: Diseño

KKOS

Integrante	LU	Correo electrónico
Burdman, Kevin	18/22	burdmankevin@gmail.com
Frau, Kenneth	189/22	kenneth.frau@gmail.com
Majic, Santiago	644/22	santiago.majic@protonmail.com
Kerbs, Octavio	64/22	octaviokerbs@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Decisiones tomadas

- 1) Si una compra de un item deja el stock de dicho item en el puesto como 0, NO remueve el item del menu del puesto.
- 2) Como el TAD no lo especifica, el precio de un producto puede ser 0.
- 3) Asumimos que la división y, como resultado, aplicar un descuento a una compra de un Item son ambas $O(1)$.
- 4) Representamos a una secuencia como una lista doblemente enlazada, de tal forma que agregar al principio/final sea $O(1)$.
- 5) Se asume que copiar un natural, al igual que comparar dos naturales, es $O(1)$.
- 6) Los arreglos se representan como vectores
- 7) El indice de los vectores empieza en 1.
- 8) Se asume que modulo y division entera son $O(1)$
- 9) Se decidio incorporar una cola de prioridad a la estructura. Esta está diseñada como modulo aparte, en la que es representada como arbol binario en la que cada nodo tiene acceso al padre junto a un elemento de mayor prioridad.
- 10) En este TP, se usan puntero(tipo de dato) y &tipo de dato de forma indiscriminada. Ambas refieren a una referencia a una variable del tipo de dato dado. De igual forma, *variable refiere al elemento al que apunta una referencia, de manera análoga a siguiente o anterior de un iterador.
- 11) Decidimos no implementar la funcion "definirRápido" para el modulo DiccLog (aplica lo mismo en Agregar Rápido para conjuntos) pues es similar a la ya conocida "definir" (agregar) de los módulos basicos pero evitando la busqueda inicial. La complejidad es similar.
- 12) No ponemos como rep que el dinero > 0 para las personas que alguna vez realizaron un gasto en los puestos, para no tener la complejidad agregada de eliminar a una persona que compro de un puesto si un hackeo le deja el gasto en 0.
- 13) Tanto el diccLog como el conjunto del módulo puesto y el modulo lollapatuza se representan con AVLs, asi la complejidad de agregar o definir, pertenencia o definido? y significado son todas de orden $O(\log(n))$
- 14) Definimos para el tipo de dato "Heap" la relacion de orden $>_{\alpha}$ segun el tipo de dato usado: por ejemplo, en el heap de gastos, se tiene como criterio un gasto mayor o un gasto igual y un id menor. En puestos, se usa el criterio de puestoid menor, definiendo asi un min heap a partir de un max heap.
- 15) Tratamos de arreglar todos los errores graficos pero hay un par que fueron imposibles de solucionar, tener en cuenta este detalle.
- 16) Tenemos en el diseño de un puesto el elemento VentasPorPersona, que, para cada persona en un puesto dado, enlista todas las compras que realizo la persona en ese puesto. Este elemento se ve afectado por la compra de un item, agregando la compra respectiva a la lista, pero no se ve alterada por un hackeo a dicho puesto, con el fin de no

complicar mucho la estructura del Lollapatuza. Notar que esto rompe con la consistencia interna del Lollapatuza y no cumple con lo pedido en el TAD con respecto al hackeo.

17) Asumimos que los diccionarios que se pasan como entrada para las funciones crearPuesto y crearLolla son diccionarios diseñados con un ABB. Dado esto, el peor caso para buscar un elemento y su significado o evaluar si tal elemento está definido, es $\mathcal{O}(n)$. Como estos diccionarios solo aparecen en las funciones que crean, que no tienen cotas de complejidad pedida, no rompen con lo pedido en el enunciado.

2. Módulo Lollapatuza

Interfaz

Usa: $\text{diccLog}(a, b)$, $\text{conj}(a)$, puesto , idpuesto , item , cant , persona , dinero , nat , bool

Se explica con: LOLLAPATUZA

géneros: lolla

Operaciones:

CREARLOLLA(**in** $\text{puestos} : \text{dicc}(\text{idpuesto}, \text{puesto})$, **in** $\text{personas} : \text{conj}(\text{persona})$)
 $\rightarrow \text{res} : \text{lolla}$
Pre $\equiv \{\text{vendenAlMismoPrecio}(\text{significados}(\text{puestos})) \wedge \text{NoVendieronAun}(\text{significados}(\text{personas})) \wedge \neg \emptyset?(\text{personas}) \wedge \neg \emptyset?(\text{claves}(\text{puestos}))\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{CrearLolla}(\text{puestos}, \text{personas})\}$
Complejidad: $\mathcal{O}(P^2 + \sum_{\text{pid} \in \text{claves}(\text{puestos})} (\text{crearPuesto}(\text{significado}(\text{pid}, \text{puestos}))) + P \cdot \log(P) + A \cdot \log(A) \cdot I \cdot \log(I) + P \cdot I \cdot \log(i))$
Descripción: Crea una instancia sin ventas de un Lollapatuza, en el cual los puestos de comida y las personas que pueden realizar acciones pertenecientes al TAD ya están dados como parámetros de entrada.
Aliasing: Los parametros de entrada se pasan como referencia no modificable, se devuelve una referencia modificable a res.

VENDER(**in/out** $\text{lp} : \text{lolla}$, **in** $\text{pi} : \text{puestoid}$, **in** $\text{p} : \text{persona}$, **in** $\text{i} : \text{item}$, **in** $\text{c} : \text{cant}$)
Pre $\equiv \{p \in \text{personas}(\text{lp}) \wedge \text{def?}(\text{pi}, \text{puestos}(\text{lp})) \wedge_L i \in \text{menu}(p) \wedge_L \text{haySuficiente}(\text{obtener}(\text{pi}, \text{puestos}(\text{lp})), c) \wedge \text{lp} =_{\text{obs}} \text{lp}_0\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{vender}(\text{lp}_0, \text{pi}, p, i, c)\}$
Complejidad: $\mathcal{O}(\log A + \log I + \log P)$
Descripción: Realiza una venta de un producto perteneciente a un puesto a una persona, aplicando, si existe tal, un descuento específico al producto y puesto dado, tal que la cantidad necesaria para poder aplicar ese descuento es menor a la cantidad comprada, pero es simultáneamente la mayor de las cantidades menores a la que se compró.
Aliasing: lp es una referencia modificable, tanto en entrada como en salida. El resto de los parametros se pasan por referencia no modificable.

HACKEAR(**in/out** $\text{lp} : \text{lolla}$, **in** $\text{p} : \text{persona}$, **in** $\text{i} : \text{item}$)
Pre $\equiv \{\text{ConsumioSinPromoEnAlgunPuesto}(\text{lp}, p, i) \wedge \text{lp} =_{\text{obs}} \text{lp}_0\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{hackear}(\text{lp}_0, p, i)\}$
Complejidad: $\mathcal{O}(\log(A) + \log(I) + \log(P))$
Descripción: Hackea una compra de un ítem consumido por una persona. Se hackea el puesto de menor ID en el que la persona haya consumido ese ítem sin promoción, aumentandole el stock del ítem en 1 y modificando el registro de la persona en sus ventas, reduciendo el gasto en el ítem del hackeo y haciendo el puesto elegido no hackeable, de ser necesario
Aliasing: lp se pasa como referencia modificable. El resto de los parametros de entrada se pasan como referencia no modificable.

GASTOTOTAL(**in** $\text{lp} : \text{lolla}$, **in** $\text{p} : \text{persona}$) $\rightarrow \text{res} : \text{dinero}$
Pre $\equiv \{p \in \text{personas}(\text{lp})\}$
Post $\equiv \{\text{res} =_{\text{obs}} \text{gastoTotal}(\text{lp}, p)\}$
Complejidad: $\mathcal{O}(\log A)$
Descripción: Obtiene el gasto total de una persona, como la suma de los gastos de la

persona en todos los puestos del lolla.

Aliasing: lp se pasa como referencia no modificable, mientras que p es no modificable y res es una copia, ya que dinero es nat.

MASGASTÓ(**in** lp: lolla) \rightarrow res : persona

Pre $\equiv \{\neg \emptyset?(personas(lp))\}$

Post $\equiv \{res =_{obs} masGasto(lp)\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Obtiene la persona que más dinero gastó en total. Si hay más de una persona que gastó el monto máximo, desempata por ID de la persona, eligiendo el de ID menor

Aliasing: El parametro de entrada se pasan como referencia no modificable, y res es una copia, ya que persona es renombre de nat. Al ser un dato de "lectura", no queremos que el usuario lo manipule manualmente. Solo se modifica mediante ventas y Hackeos, por lo que devuelve una referencia no modificable.

MENORSTOCK(**in** lp: lolla, **in** i: item) \rightarrow res : puesto

Pre $\equiv \{i \in dameUno(menu(obtenerUno(puestos)))\}$

Post $\equiv \{res =_{obs} menorStock(lp, i)\}$

Complejidad: $\mathcal{O}(P * (\log(P) + \log(I)))$

Descripción: Obtiene el puesto de menor stock para un ítem dado. Si hay más de un puesto que tiene stock mínimo, o no hay ningun puesto que venda el ítem, devuelve el de menor ID.

Aliasing: Los parametros de entrada se pasan como referencia no modificable, y res es una referencia no modificable.

PERSONAS(**in** lp: lolla) \rightarrow res : conj(persona)

Pre $\equiv \{\text{True}\}$

Post $\equiv \{res =_{obs} personas(lp)\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve todas las personas que pueden formar parte en las distintas acciones del Lollapatuza

Aliasing: lp se pasa como referencia no modificable, res se devuelve como una referencia no modificable

PUESTOS(**in** lp: lolla) \rightarrow res : dicc(puestoid, puesto)

Pre $\equiv \{\text{True}\}$

Post $\equiv \{res =_{obs} puestos(lp)\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Obtiene los puestos de comidas que pueden formar parte de las acciones del Lollapatuza, estando asociados cada uno con un ID único.

Aliasing: lp se pasa como referencia no modificable, res se devuelve como una referencia no modificable

Auxiliares

$>_{\alpha}(\text{in } personaConGasto1: \text{tupla } \langle \text{dinero}, \text{persona} \rangle, \text{in } personaConGasto2: \text{tupla } \langle \text{dinero}, \text{persona} \rangle) \rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{res =_{obs} personaConGasto1 >_{\alpha} personaConGasto2\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Compara dos tuplas del tipo dado y devuelve true sii la primera es mayor a la segunda según el criterio del TP

Aliasing: Los parámetros de entrada se pasan por referencia no modificable

$>_{\alpha}(\text{in } puestoid1: \text{puestoid}, \text{in } puestoid2: \text{puestoid}) \rightarrow res : \text{bool}$

Pre $\equiv \{\text{true}\}$

Post $\equiv \{\text{res} =_{\text{obs}} \text{puestoid1} >_{\alpha} \text{puestoid2}\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Comparacion de puestoid dado el criterio de este tp, en el que un puestoid tiene mayor prioridad que otro si el primero es menor(ambos son naturales)

Aliasing: Los parámetros de entrada se pasan por referencia no modificable

Representación

Representación de Lollapatuza:

lolla se representa con estr

```
donde estr es tupla( ColaDeGastos: colaPrioridad(tupla<dinero, persona>),
, personas: diccLog(persona, tupla(enGastos*: itColaPrioridad,
comprasSinDescuentoPorItem: diccLog(item,
tupla(colaPrioridad(comprasSinDescuentoEnPuesto), diccLog(puestoid,
itColaPrioridad*)))
, conjPer: conj(persona)
, puestos: conj(puesto))
, puestosPorId: diccLog(puestoid, puesto)
, precio: dicclog(item, nat))
```

donde **item**, **precio**, **persona**, **dinero**, **puestoid** son NAT

donde **comprasSinDescuentoEnPuesto** es Tupla (puntero(puesto), puestoid, cantidad
)

Decisiones Tomadas de la Representacion:

(1) ColaDeGastos: Una cola de tupla gasto-persona. La cola esta ordenada por los gastos. Esta se modifica cuando hay una venta o un hackeo.

(2) personas:

Por un lado, un puntero a su posicion en la cola de gastos, para poder accederla y modificarla en $O(1)$ una vez que se encuentra a la persona. Por otro lado, cada persona tiene, para todo item que puede ser vendido, una tupla, donde el primer elemento es una cola de prioridad en la que se ordenan, por menor id de puesto, todos los puestos en los que la persona tiene registrada al menos una compra sin descuento del item dado. Por otro lado, a la hora de buscar un puesto que tiene una compra sin descuento, para no tener que buscar en $O(P)$, usamos un diccLog donde las claves son ids de puestos, y los significados son punteros a las posiciones en la cola de prioridad (asi podemos buscar un puesto en $\log(P)$ a la hora de modificar un puesto). Finalmente, cada elemento de la cola de prioridad tiene un puntero al puesto(así se puede acceder al puesto en $O(1)$), el puestoid del puesto(así podemos establecer la relacion de orden del heap) y la cantidad del item comprado sin descuento(así la aumentamos cuando compramos sin descuento el item, y, cuando el contador llega a 0 despues de un hackeo, desencolamos el puesto y este pasa a no ser hackeable(y tambien borramos la clave del diccLog))

(3) conjPer: el conjunto de personas que estan en el Lollapatuza.

(4) puestos: el conjunto de puestos que estan en el Lollapatuza.

(5) puestosPorId: dado un puestoID nos devuelve el puesto.

(6) precio: dado un item nos devuelve su precio.

(7) En dos ocasiones de esta estructura, aparecen dos estructuras relacionadas: una cola de prioridad y un diccionario donde todos los significados apuntan a un nodo de dicha cola de prioridad. La cola de prioridad existe porque, tanto para los gastos de una persona como para los descuentos de un item dado, existe un sistema de prioridad en los que no se puede tener en cuenta solo el elemento de mayor prioridad: en gastosPorPersona, se necesita saber quien es la segunda persona que mas gastó dado un caso de un hackeo que le baja el gasto, y para comprasSinDescuentoPorItem se necesita saber cual el siguiente puesto a hackear una vez que un puesto no es hackeable: esto se logra o bien desencolando o bien haciendo un Sift Down. Hay casos muy particulares en los cuales necesito buscar un nodo de la cola de prioridad que no es el de mayor prioridad: en el caso de

gastosPorPersona, es simplemente actualizar el gasto de una persona luego de una venta, y en el de ComprasSinDescuentoPorItem, cuando necesito registrar una compra sin descuento, necesito registrar si tal puesto ya está registrado o no. Como buscar en un heap es $O(n)$ (y eso no está permitido en las cotas del tp) necesito bajar la complejidad de la búsqueda a, a lo sumo, orden $O(\log(n))$. Para eso, usamos un Diccionario Logarítmico, cuyo significado apunta al nodo del heap, así podemos reducir la búsqueda del elemento a $O(\log(n))$

Invariante de Representacion

- 1) $e.conjPer = claves(e.personas)$.
- 2) $e.puestos = significados(e.puestosPorId)$.
- 3) Para todo item que es clave de precio, existe un puesto que vende tal item.
- 4) Para todo item que es clave de precio, todos los puestos que venden ese item lo venden al significado del item en precio.
- 5) Además en el $diccLog(puestoid, itColaPrioridad)$ de las compras de un item para toda persona, todos los $puestosid$ pertenecen a $claves(puestosPorId)$.
- 6) Un puesto tiene un único $puestoid$ que lo identifica.
- 7) Todos los nodos de $ColaDeGastos$ representan a una persona de personas, con su respectivo gasto total. Toda persona en $personas$ tiene un nodo en $ColaDeGastos$.
- 8) No hay dos nodos con la misma persona en $ColaDeGastos$.
- 9) Para toda persona registrada, el puntero en $Gastos$ apunta al nodo de la persona en $ColaDeGastos$ (el segundo elemento de la tupla coincide).
- 10) Para toda persona, $claves(comprasSinDescuentoPorItem) = claves(precio)$ (decisión: definimos todos los items para el diccionario de $comprasSinDescuentoPorItem$ al momento de crear una instancia del Lollapatuza, para no chequear si existe el item al hacer una compra).
- 11) Para toda persona, en $comprasSinDescuentoPorItem$, un puntero es significado de un $puestoid$ en el diccionario sii en el nodo a donde apunta, el $puestoid$ del nodo coincide con el $puestoid$ del puesto (es decir, si se busca el puesto por su $puestoid$ en el $puestosPorID$, debería apuntar al mismo puesto que el puntero del nodo).
- 12) Todo nodo asociado a un puesto de la cola de prioridad de $comprasSinDescuentoPorItem$ para toda persona dada, cumple que la cantidad del item comprado sin descuento que figura en el nodo es menor o igual a la cantidad del item comprada sin descuento que figura en la lista de ventas del puesto del nodo. (Es decir, vas al puesto que figura, buscas la persona dada y, dentro de las ventas asociadas a esa persona, filtras por las compras sin descuento del item dado. La suma de las cantidades de todas esas compras es mayor o igual a la cantidad sin descuento que figura en el nodo)
- 13) Para toda persona, su gasto total es menor o igual a la suma de los gastos de la persona en todos los puestos del conjunto de puestos. (Es decir, si sumas todas las ventas de una persona en todos los puestos del lollapatuza, el total debe ser mayor o igual al gasto total de la persona (ya que el hackeo rompe la consistencia))
- 14) En $comprasSinDescuentoPorItem$, para todo pid definido en el diccionario Logaritmico de la tupla, su significado apunta a un nodo de la cola de prioridad de la misma tupla, en la que el $puestoid$ de la tupla del nodo coincide con el $puestoid$ que usamos para la busqueda
- 15) Para todo nodo de la cola de prioridad de $comprasSinDescuentoPorItem$, el puesto es hackeable, es decir, la cantidad que se compro sin descuento del item en ese puesto es mayor a 0.
- 16) Hay, para todo nodo en la $colaDePrioridad$ en $comprasSinDescuentoPorItem$, un único $puestoid$ en las claves del diccionario Logaritmico de la misma tupla cuyo significado apunta a ese nodo. (Es decir, hay una relacion 1 a 1 entre los $puestosid$ de los nodos y las claves de el diccionario logaritmico)

Rep : $dic \rightarrow bool$

$Rep(d) \equiv true \iff (1 \wedge 2) \wedge_L (5 \wedge 10) \wedge_L 9 \wedge 11 \wedge_L 3 \wedge 4 \wedge 6 \wedge 7 \wedge 8 \wedge 12 \wedge 13 \wedge 14 \wedge 15 \wedge$
16

Abs : estr $e \longrightarrow$ lollapatuza $\{\text{Rep}(e)\}$

Abs(e) $\equiv (\forall l: \text{lolla}) (l =_{obs} e \iff$
 $\text{puestos}(l) =_{obs} e.\text{puestosPorId} \wedge$
 $\text{personas}(l) =_{obs} e.\text{conjPer})$

(1) $\{ e.\text{conjper} = \text{claves}(e.\text{personas}) \}$

(2) $\{ (\forall p: \text{puesto}) (p \in e.\text{puestos} \iff (\exists \text{pid}: \text{puestoid}) (\text{def?}(\text{pid}, e.\text{puestosPorId}) \wedge_L$
 $\text{obtener}(\text{pid}, e.\text{puestosPorId}) = p)) \}$

(3) $\{ (\forall i: \text{item}) (i \in \text{claves}(e.\text{precios}) \iff (\exists \text{pid}: \text{puestoid}) (\text{def?}(\text{pid}, e.\text{puestosPorId})$
 $\wedge_L i \in \text{claves}(\text{obtener}(\text{pid}, e.\text{puestosPorId}).\text{menu})))$

}

(4) $\{ (\forall i: \text{item}) (\text{def?}(i, e.\text{precios}) \Rightarrow_L ((\forall \text{pid}: \text{puestoid}) (\text{def?}(\text{puestoid}, e.\text{puestosPorId})$
 $\wedge_L i \in \text{claves}((\text{obtener}(\text{puestoid}, e.\text{puestosPorId}).\text{menu}) \Rightarrow_L$
 $(\text{obtener}(i, \text{obtener}(\text{puestoid}, e.\text{puestosPorId}).\text{menu}).\text{precio} = \text{obtener}(i, e.\text{precios})))$

}

(5) $\{ (\forall p: \text{persona}) (\text{def?}(p, \text{personas}) \Rightarrow_L$
 $((\forall i: \text{item}) (\text{def?}(i, \text{obtener}(p, \text{personas}).\text{comprasSinDescuentoPorItem}))) \Rightarrow_L$
 $\text{claves}(\pi_2(\text{obtener}(i, \text{obtener}(p, \text{personas}).\text{comprasSinDescuentoPorItem}))) \subseteq \text{claves}(\text{puestosPorId})$

}

(6) $\{ (\forall \text{pid}, \text{pid}': \text{puestoid}) ((\text{pid}, \text{pid}' \subseteq \text{claves}(e.\text{puestosPorId}) \wedge \text{pid} \neq \text{pid}') \Rightarrow_L$
 $\text{obtener}(\text{pid}, e.\text{puestosPorId}) \neq \text{obtener}(\text{pid}', e.\text{puestosId})) \}$

(7) $\{ (\forall p: \text{persona}) (\text{def?}(p, e.\text{personas}) \iff (\exists t: \text{tupla}(\text{dinero}, \text{persona})) (t \in \text{colaA-}$
 $\text{Conjunto}(e.\text{ColaDeGastos}) \wedge \pi_2(t) = p) \}$

colaAConjunto : colaPrioridad(α) \longrightarrow conj(α)

colaAConjunto($c : \text{colaPrioridad}(\alpha)$)) { **if** vacía?(c) **then** \emptyset
else Ag(proximo(c), colaAConjunto(desencolar(c))) **end if** }

(8) $\{ (\forall p, p': \text{persona}) (\text{def?}(p, \text{personas}) \wedge \text{def?}(p', \text{personas}) \wedge (p \neq p' \iff ((\text{obte-}$
 $\text{ner}(p, e.\text{personas}).\text{enGastos} \neq \text{obtener}(p', e.\text{personas}).\text{enGastos})) \}$

(9) $\{ (\forall p: \text{persona}) (\text{def?}(p, \text{personas}) \Rightarrow_L \pi_2(*(\text{obtener}(p, e.\text{personas}).\text{enGastos})) =$
 $p)$

}

(10) $\{ (\forall p: \text{persona}) (\text{def?}(p, e.\text{personas}) \Rightarrow_L$
 $\text{claves}(\text{obtener}(p, e.\text{personas}).\text{comprasSinDescuentoPorItem})) = \text{claves}(e.\text{precio})$

}

(11) $\{ (\forall p: \text{persona}) (\forall i: \text{item}) (\text{def?}(p, e.\text{personas})) \wedge_L$
 $\text{def?}(\text{obtener}(p, \text{personas}).\text{comprasSinDescuentoPorItem}, i)$
 $\Rightarrow_L * \text{obtener}(\text{obtener}(p, e.\text{personas}).\text{comprasSinDescuentoPorItem}, i).\text{comprasSinDescuentoEnPuesto.puesto}$
 $= \text{obtener}(\text{puestosPorId}, \text{puestoid}) \}$

(12) $\{ (\forall p: \text{persona}) (\text{def?}(p, e.\text{personas}))$
 $\Rightarrow_L ((\forall i: \text{item}) (\text{def?}(i, \text{obtener}(p, e.\text{personas}).\text{comprasSinDescuentoPorItem}))$
 $\Rightarrow_L ((\forall t: \text{tupla}(\text{puntero}(\text{puesto}), \text{puestoid}, \text{cantidad}))$
 $(t \in \pi_1(\text{obtener}(i, \text{obtener}(p, e.\text{personas}).\text{comprasSinDescuentoPorItem}))$
 $\Rightarrow_L \pi_3(\text{obtener}(i, \text{obtener}(p, e.\text{personas}).\text{comprasSinDescuentoPorItem}).\text{cantidad}$
 $\leq \text{cantidadTotalCompradaSinDescuentoDelItemDadoEnElPuestoDado}(p, i, \text{pid})))) \}$

(13) $\{ (\forall a: \text{persona}) (\text{def?}(a, e.\text{personas}) \Rightarrow_L (\forall p: \text{puesto}) ($
 $\sum_{p \in \text{claves}(e.\text{puestosPorId})} (\text{if } \text{def?}(\text{obtener}(\text{obtener}(e.\text{puestosPorId}, p).\text{ventasPorPersona},$
 $a))$

```

then obtener(obtener(e.puestosPorid, p).ventasPorPersona, a).gastoTotal else 0)
 $\geq$  obtener(e.personas, a).enGastos.dinero ) }

(14) { (( $\forall$  p:persona)(def?(p,e.personas))
 $\Rightarrow_L$  (( $\forall$  i:item)(def?(i,obtener(p,e.personas).comprasSinDescuentoPorItem)
 $\Rightarrow_L$  (( $\forall$  t:tupla  $\langle$ puntero(puesto),puestoid,cantidad $\rangle$ )
(t  $\in \pi_1$ (obtener(i,obtener(p,e.personas).comprasSinDescuentoPorItem))
 $\Rightarrow_L$   $\pi_3$ (obtener(i,obtener(p,e.personas).comprasSinDescuentoPorItem)).cantidad  $\leq$ 
cantidadTotalCompradaSinDescuentoDelItemDadoEnElPuestoDado(p,i,pid)))))) }

(15) { (( $\forall$  p:persona)(def?(p,e.personas))
 $\Rightarrow_L$  (( $\forall$  i:item)(def?(i,obtener(p,e.personas).comprasSinDescuentoPorItem)
 $\Rightarrow_L$  (( $\forall$  t:tupla  $\langle$ puntero(puesto),puestoid,cantidad $\rangle$ )
(t  $\in \pi_1$ (obtener(i,obtener(p,e.personas).comprasSinDescuentoPorItem))
 $\Rightarrow_L$   $\pi_3$ (obtener(i,obtener(p,e.personas).comprasSinDescuentoPorItem))  $> 0$ )))))) }

(16) { (( $\forall$  p:persona)(def?(p,e.personas))
 $\Rightarrow_L$  (( $\forall$  i:item)(def?(i,obtener(p,e.personas).comprasSinDescuentoPorItem)
 $\Rightarrow_L$  (( $\forall$  pid,pid':puestoid) pid,pid'  $\in$  claves( $\pi_2$ (obtener(i,obtener(p,e.personas).comprasSinDescuentoPorItem)))
 $\wedge$  pid  $\neq$  pid'
 $\Rightarrow_L$  obtener(pid, $\pi_2$ (obtener(i,obtener(p,e.personas).comprasSinDescuentoPorItem)))
 $\neq$  obtener(pid', $\pi_2$ (obtener(i,obtener(p,e.personas).comprasSinDescuentoPorItem))))))
}

```

Algoritmos

iCREARLOLLA(in *puestos*: dicc(idpuesto, puesto), in *personas*: conj(persona)) \rightarrow *res*
: *lolla*

```
1: res  $\leftarrow$  (Vacio(),Vacio(),Vacio(),Vacio(),Vacio(),Vacio())
2: it  $\leftarrow$  crearIt(puestos)
3: while haySiguiente(it) do
4:   viejoPuesto  $\leftarrow$  significado(puestos,siguiente(it))
5:   nuevoPuesto  $\leftarrow$  crearPuesto(precio(viejoPuesto),stock(viejoPuesto),descuentos(viejoPuesto))
6:   definirRapido(res.puestosPorId,siguiente(it),nuevoPuesto)
7:   AgregarRapido(res.puestos,nuevoPuesto)
8:   itItem  $\leftarrow$  crearIt(menu(viejoPuesto))
9:   while haySiguiente(itItem) do
10:    definir(res.precio,siguiente(itItem),precio(viejoPuesto,siguiente(itItem)))
11:    Avanzar(itItem)
12:   end while
13:   Avanzar(it)
14: end while
15: itper  $\leftarrow$  CrearIt(personas)
16: while haySiguiente(itper) do
17:   AgregarRapido(res.conjPer,siguiente(itper))
18:   definirRapido(res.personas,siguiente(itper),⟨nullptr,Vacio()⟩)
19:   dineroGastado  $\leftarrow$  ⟨0,siguiente(itper)⟩
20:   punteroEnGasto  $\leftarrow$  encolar(res.ColaDeGastos, siguiente(itper))
21:   persona  $\leftarrow$  significado(res.personas,siguiente(itper))
22:   persona.enGastos  $\leftarrow$  punteroEnGasto
23:   itItem2  $\leftarrow$  crearIt(res.precio)
24:   while haySiguiente(itItem2) do
25:    definirRapido(persona.comprasSinDescuentoPorItem, siguiente(itItem2),⟨Vacía(),Vacio()⟩)
26:    Avanzar(itItem2)
27:   end while
28:   Avanzar(itper)
29: end while
30: return res
```

Complejidad: $\mathcal{O}(P^2 + \sum_{pid \in puestos} (\text{crearPuesto}(\text{significado}(pid, puestos))) + P \cdot \log(P) + A \cdot \log(A) \cdot I \cdot \log(I) + (P \cdot I \cdot \log(I)))$

Justificación: Asumiendo que los diccionarios de entrada estan diseñados como ABB, en peor caso buscar en el conjunto o buscar su significado es $\mathcal{O}(n)$, por lo que, si lo hacemos para cada puesto, se encuentra en orden de $\mathcal{O}(n)$. Para cada puesto en el diccionario, debemos crear una instancia del tipo puesto. Crear puesto es $\mathcal{O}(I^2 + \sum_{i' \in i} (\#DescuentosDelItem_{i'}^2 + MaximaCantConDescuento^2) + I \cdot \log(I))$ Definir rapido es $\mathcal{O}(\log(n))$, con $0 < n \leq P$, lo que se aproxima, al sumar los logaritmos, a $P \cdot \log(P)$ (Stirling). Agregar rapido es $\mathcal{O}(\log(n))$ pues es un conjunto logaritmico. Como las claves del conjunto empiezan en 0 y se agregan progresivamente elementos. Agregar todo puesto es $\mathcal{O}(P \cdot \log(P))$ (Stirling). Para toda persona en *personas*, se agrega al conjunto de personas y se define en el diccionario de personas. Como agregar/definir es $\mathcal{O}(\log(n))$, con $0 < n \leq a$, por Stirling, esto es $\mathcal{O}(A \cdot \log(A))$, pero tambien para cada persona definimos automaticamente todos los items para *comprasSinDescuentoPorItem* con todo vacio, lo cual, usando la misma logica, es $\mathcal{O}(I \cdot \log(I))$. Es decir, todo este proceso es $\mathcal{O}(A \cdot \log(A) \cdot I \cdot \log(I))$. En peor caso, todos los puestos tienen el mismo menu, que consiste de todos los items posibles. En este caso, defino en precio 1 vez para cada item, lo cual, por Stirling, es $\mathcal{O}(I \cdot \log(I))$, y para todo puesto restante, debemos redefinir todos los items con el mismo precio (por precondition), por lo que, ahora con todos los items, la complejidad para definir es $\mathcal{O}(\log(i))$. Por ende, esta parte tiene complejidad $\mathcal{O}((P \cdot I \cdot (\log(I)))$ Toda otra operacion es $\mathcal{O}(1)$.

Resumen: Primero crea el esqueleto de la estructura, completamente vacío. Itera por cada puesto en los puestos de entrada, definimos un puesto de la estructura usando `crearPuesto`, para luego definirlo en los puestos en el diccionario, asociado al `puetoid` que lo asocia en el diccionario de entrada, y además agrega el puesto al conjunto de puestos. Luego, para cada item que vende el puesto, busca su precio(se asume que un item tiene igual precio en todos los puestos), y lo define en el diccionario item-precio de la estructura(notar que acá no usa `definir rapido`, ya que puede haber items compartidos entre los puestos). Luego de iterar sobre los puestos, itera sobre las personas, agregando cada una al conjunto de personas y definiendolo en el diccionario de personas, definiendo como su significado un `nullptr` y un diccionario de items vacío. Define su nodo en la cola de prioridad de gastos, inicializando su gasto a 0, y lo encola. Como `encolar` devuelve un puntero al nodo encolado, guardo este puntero y lo seteo como un acceso en $O(1)$ a la posición en la cola de gastos, para poder modificarlo en tiempo constante. Finalmente, para cada item vendido en Lollapatoosa, define las compras sin descuento por item para cada item para la persona dada, que, dada la estructura, es una cola de prioridad vacía y un diccionario vacío.

```

iVENDER(in/out  $e$ : estr, in  $pi$ : puestoid, in  $a$ : persona, in  $i$ : item, in  $c$ : cant)
1: puesto  $\leftarrow$  significado( $e$ .puestosPorId,  $pi$ )
2: per  $\leftarrow$  significado( $e$ .personas,  $a$ )
3: nuevaCompra  $\leftarrow$  AgregarVenta(puesto,per,i,c)
4:  $\pi_1(*(\text{per.enGastos})) \leftarrow \pi_1(*(\text{per.enGastos})) + \pi_2(\text{nuevaCompra})$ 
   nuevaCompra es una tupla  $\langle \text{puntero}(\text{puesto}), \text{precio}, \text{cantidad} \rangle$ 
5: siftUp( $e$ .ColaDeGastos,  $*(\text{per.enGastos})$ )
6: if  $\pi_1(\text{nuevaCompra}) \neq \text{nullptr}$  then (Si es nullptr, la compra es con descuento)
7:   nuevaCompraCopia  $\leftarrow \langle \pi_1(\text{nuevaCompra}), pi, \pi_3(\text{nuevaCompra}) \rangle$ 
   nuevaCompraCopia es una tupla  $\langle \text{puntero}(\text{puesto}), \text{puestoid}, \text{cantidad} \rangle$ 
8:   itemSinDescuento  $\leftarrow$  significado(per.comprasSinDescuentoPorItem,i)
9:   if definido?( $\pi_2(\text{itemSinDescuento}), pi$ ) then
10:     $*(\text{significado}(\pi_2(\text{itemSinDescuento}), pi)) \leftarrow *( \text{significado}(\pi_2(\text{itemSinDescuento}), pi)) +$ 
     $\pi_3(\text{nuevaCompraCopia})$ 
11:  else
12:    it  $\leftarrow$  encolar( $\pi_1(\text{itemSinDescuento}), \text{nuevaCompraCopia}$ )
13:    definirRapido( $\pi_2(\text{itemSinDescuento}), pi, it$ )
14:  end if
15: end if

```

Complejidad: $\mathcal{O}(\log(P) + \log(I) + \log(A))$

Justificacion: Primero obtenemos la persona del diccionario logarítmico de personas, y el puesto del diccionario logarítmico de puestos ($\mathcal{O}(\log(P) + \log(A))$). Llama a AgregarVenta, cuya complejidad en peor caso es de ($\mathcal{O}(\log(I) + \log(A))$). Para actualizar el gasto de la persona accede al puntero a la cola de prioridad y hace un SiftUp ($\mathcal{O}(\log(P))$) y, en el caso de que la compra no tenga descuentos, el peor caso es que tenga que definir un nuevo puesto en el que la compra del item no tenga descuento(en otro caso busco el puntero al nodo en la cola de prioridad en el diccionario y agrega adelante la cantidad a la tupla), ya que busca el elemento en el diccionario y no lo encuentra, lo encola en la cola de prioridad y inserta un puntero al nodo de dicha cola en el diccLog. El orden de esto ultimo es $O(3*\log(P)) = O(\log(P))$. Sumando todas las complejidades nos queda esta complejidad.

Resumen:

Primero busca el puesto por su id en puestosPorID y la la persona en personas para poder actualizar ambos con respecto a la nueva compra.

Al llamar a nueva compra, agrega la compra a la lista de compras de la persona en el puesto dado, y resta el stock a la cantidad dada.

Esta funcion devuelve una tupla $\langle \text{puntero}(\text{puesto}), \text{precio}, \text{cantidad} \rangle$ que registra todo lo que necesitamos para actualizar la persona: en el nodo de la persona de la cola de prioridad por el gasto total, le suma el precio y llama a siftUp de esa cola de prioridad, que le aumenta la prioridad a la persona de ser necesario. Luego evalua el puntero al puesto: si es nullptr, por como funciona agregar venta, implica que la compra fue con descuento, por lo que no necesito hacer nada más.

De ser una compra sin descuento, agregarVenta devuelve un puntero al puesto para poder modificarlo en caso de hackeo. En este ultimo caso, debe agregar la compra a las compras sin descuento por item, asociado al puesto dado. Tenemos aquí dos casos: el puesto ya aparece en la cola de prioridad, en cuyo caso solo accedemos a su posicion mediante la busqueda en diccionario por su puestoid y le aumentamos la cantidad comprada sin descuento en la cantidad de la compra. El otro caso es que no tenemos un nodo asociado al puesto en la cola de prioridad, en cuyo caso debemos encolarlo (para eso creamos la tupla nuevaCompraCopia). Como encolar devuelve un puntero al nodo asociado al puesto, lo ultimo que hacemos es definirlo en el diccionario que, en su significado, tiene un puntero al nodo, así podemos acceder a este en $O(\log(P))$.

iHACKEAR(in/out e : estr, in a : persona, in i : item)

```
1: per  $\leftarrow$  significado( $e$ .personas, $a$ )
2: precioDelItem  $\leftarrow$  buscar( $e$ .precio,item)
3: ComprasSinDescuento  $\leftarrow$  significado(per.comprasSinDescuentoPorItem, $i$ )
4: PuestoDeCompraABorrar  $\leftarrow$  proximo( $\pi_1$ (ComprasSinDescuento))
5: pid  $\leftarrow$   $\pi_2$ (PuestoDeCompraABorrar)
6: EliminarVentaSinDescuento( $\pi_1$ (PuestoDeCompraABorrar)), $a$ , $i$ ,precioDelItem)
7:  $\pi_3$ (PuestoDeCompraABorrar)  $\leftarrow$   $\pi_3$ (PuestoDeCompraABorrar) - 1
8: if  $\pi_3$ (PuestoDeCompraABorrar) = 0 then
9:   desencolar( $\pi_1$ (ComprasSinDescuento))
10:  borrar( $\pi_2$ (ComprasSinDescuento), pid)
11: end if
12: gastoDePersona  $\leftarrow$  *(per.enGastos)
13:  $\pi_1$ (gastoDePersona)  $\leftarrow$   $\pi_1$ (gastoDePersona) - precioDelItem
14: siftDown( $e$ .ColaDeGastos,gastoDePersona)
```

Complejidad: $\mathcal{O}(\log(P) + \log(A) + \log(I))$

Justificacion: Hay operaciones del algoritmo que refieren a manipulación y eliminacion de punteros, todos $\mathcal{O}(1)$. Las unicas operaciones que superan esto refieren a busquedas en diccLogs, en donde la cantidad de claves están acotadas por la cantidad de items, de puestos o de personas, e insercion o eliminacion de elementos en colas de prioridad, donde la cantidad de claves tambien esta acotada por la cantidad de personas o de puestos. Esto cumple la complejidad porque, en el caso de que un puesto no sea hackeable luego de la funcion, remueve el elemento de mayor prioridad del heap ($\mathcal{O}(\log(P))$) y esa es la unica operacion en la que comparamos varios puestos, y sumamos la complejidad de las operaciones de los otros casos, que, como consisten de busquedas en AVLs de Personas o items y un SiftDown del maximo de un heap de personas, es $\mathcal{O}(\log(P) + \log(A))$

Resumen: Primero buscamos la persona dentro del lollapatooza y el precio del item. Luego se busca el puesto de menor id que tenga al menos una compra sin descuento del item dado (que se sabe que existe ya que la cantidad dentro de la tupla de la cola de prioridad es mayor a 0), y por eso se accede comprasSinDescuentoPorItem de la persona dada. Dado este puesto, llama a la funcion EliminarVentaSinDescuento, que lo unico que hace es agregar 1 al stock del item, y reducir el gasto de la persona en el puesto en el precio del item (por esta razon tenemos un puntero al puesto acá, ya que esto nos permite tener acceso al puesto en $\mathcal{O}(1)$ y respetar la complejidad pedida en el caso normal). Luego reduce la cantidad hackeable del item dado (y el puesto dado) en 1 y luego chequea si el puesto ya no es hackeable, es decir, que la cantidad hackeable bajo a 0. En ese caso, debemos desencolar el heap, y, para mantener la consistencia de la estructura, borrar la referencia al nodo correspondiente en el diccionario asociado. Por esta razon es que el 2do elemento de la tupla del nodo es el puestoid, ya que nos permite borrar el puesto del diccionario en $\mathcal{O}(\log(P))$. Finalmente, se resta el precio del item al gasto total de la persona y, de ser necesario, la baja la prioridad en GastosPorPersona mediante el llamado a SiftDown. (Notar que esta función rompe la consistencia del lollapatuza y no cumple ciertas cosas del TAD, con respecto a como afecta el hackeo al puesto y sus ventas. Para más información ver el resumen de EliminarVentaSinDescuento en el módulo de Puesto de Comida)

iGASTOTOTAL(in/out e : **estr**, in a : **persona**) $\rightarrow res$: dinero

- 1: $infoPersona \leftarrow significado(e.personas, a)$
- 2: $itAlGasto \leftarrow infoPersona.enGastos$
- 3: $res \leftarrow \pi_1(*itAlGasto)$
- 4: return res

Complejidad: $\mathcal{O}(\log A)$

Justificacion: Para buscar el gasto total buscamos a la persona en el diccionario de la estructura y accedemos a su gasto total en la tupla, $\mathcal{O}(\log A) + \mathcal{O}(1)$ siendo A la cantidad de personas. Luego se accede a la cola de prioridad con el puntero al espacio de memoria que apunta al nodo asociado a la persona, lo que es $\mathcal{O}(1)$.

iMASGASTÓ(in/out l : **lolla**) $\rightarrow res$: persona

- 1: $res \leftarrow \pi_2(proximo(e.ColaDeGastos))$

Complejidad: $\mathcal{O}(1)$

Justificacion: Accedo al próximo de "colaDeGastos" en la estructura, y eso cuesta $\mathcal{O}(1)$.

iMENORSTOCK(in/out $e: \text{estr}, \text{in } i: \text{item}$) $\rightarrow res: \text{puesto}$

```

1:  $it \leftarrow \text{crearIt}(e.\text{puestosPorId})$ 
2:  $\text{puestoActual} \leftarrow \text{significado}(e.\text{puestosPorID}, \text{siguiente}(it))$ 
3:  $\text{minStock} \leftarrow \text{puestoActual}$ 
4:  $\text{idMin} \leftarrow \text{siguiente}(it)$ 
5:  $\text{Avanzar}(it)$ 
6: while  $\text{haySiguiente}(it)$  do
7:    $\text{puestoActual} \leftarrow \text{significado}(e.\text{puestosPorId}, \text{siguiente}(it))$ 
8:   if  $\text{definido?}(\text{puestoActual}.\text{menu}, i)$  then
9:      $\text{stockDelItem} \leftarrow \text{Stock}(\text{puestoActual}, i)$ 
10:    if  $(\neg \text{definido?}(\text{minStock}.\text{menu}, i) \wedge \neg \text{definido?}(\text{puestoActual}.\text{menu}, i) \wedge$ 
11:  $\text{idMin} > \text{siguiente}(it)) \vee_L (\neg \text{definido?}(\text{minStock}.\text{menu}, i) \wedge \text{definido?}(\text{puestoActual}.\text{menu}, i))$ 
12:  $\vee_L (\text{definido?}(\text{minStock}.\text{menu}, i) \wedge (\text{Stock}(\text{puestoActual}, i) > \text{Stock}(\text{minStock}, i) \vee$ 
13:  $((\text{Stock}(\text{minStock}, i) = \text{Stock}(\text{puestoActual}, i) \wedge \text{idMin} > \text{siguiente}(it))))$  then
14:    $\text{minStock} \leftarrow \text{puestoActual}$ 
15:    $\text{idMin} \leftarrow \text{siguiente}(it)$ 
16:   end if
17: end if
18:  $\text{Avanzar}(it)$ 
19: end while
20:  $res \leftarrow \text{idMin}$ 
21: return  $res$ 

```

Complejidad: $\mathcal{O}(P * (\log(P) + \log(I)))$

Justificacion: Fuera de definir y avanzar iteradores ($\mathcal{O}(1)$), para todo puesto definido, debo primero ver si el item existe en el menú ($\mathcal{O}(\log(I))$), y luego ver si esta el item en el menu del minimo anterior ($\mathcal{O}(\log(I))$), y en caso de estar definido en ambos, comparar el Stock y, en peor caso, tambien el id, ambos de acceso $\mathcal{O}(1)$ una vez que accedemos al puesto. Como tambien tenemos que buscar el puesto en puestosPorID, esto suma $\mathcal{O}(\log(P))$ a la complejidad de cada iteracion. Como se debe hacer esto para todo puesto, nos queda como complejidad $\mathcal{O}(P * (\log(P) + \log(I)))$

Resumen: Primero defino como el puesto de menor stock como el primero que aparece cuando itero por todos los puestos, sin importar si tiene el item en el menu o no. Luego itera por todos los puestos restantes, y decide si el siguiente puesto es el de menor stock segun varias condiciones: si el item no esta en el menú de ambos pero el puestoid del siguiente puesto es menor al de menorStock, actualiza el de menor id como el siguiente (esto es lo que se pide en el TP: de no haber puestos que tengan stock de ese item, devolver el de menor id. De estar definido el menu en el item, sin importar de que su stock sea 0 o que tenga mayor puestoid, ese será el mínimo). Otro caso en el que se actualiza el de menor stock es aquel en el que el siguiente puesto tiene al item en su menu y el supuesto mínimo no lo tiene. La otra es la condicion normal: el item está en el menu de ambos puestos y el stock del siguiente es menor al del minimo registrado(o la de ambos es igual, en cuyo caso se elige al de id menor). Durante el ciclo, se guardan dos variables que se actualizan: el puesto del stock minimo, y el id del puesto de stock minimo, ya que se necesita el ultimo para comparar los puestos por id en caso de desempate, y se debe devolver el primero.

iPERSONAS(in/out $e: \text{estr}$) $\rightarrow res: \text{conj}(\text{persona})$

1: $res \leftarrow e.\text{personas}$

Complejidad: $\mathcal{O}(1)$

Justificacion: Devuelve una referencia a la que se tiene acceso directo en la estructura.

iPUESTOS(in/out $e : \mathbf{estr}$) $\rightarrow res : \text{dicc}(\text{puestoid}, \text{puesto})$

1: $res \leftarrow e.\text{puestosPorId}$

Complejidad: $\mathcal{O}(1)$

Justificación: Devuelve una referencia a la que se tiene acceso directo en la estructura.

Auxiliares

$>_{\alpha}$ (**in** *personaConGasto1* : *tupla*(dinero, persona),
in *personaConGasto2* : *tupla*(dinero, persona)) $\rightarrow res$: bool

1: $res \leftarrow personaConGasto1.dinero > personaConGasto2.dinero \vee (personaConGasto1.dinero = personaConGasto2.dinero \wedge personaConGasto1.persona < personaConGasto2.persona)$

2: return res

Complejidad: $\mathcal{O}(1)$

Justificacion: En peor caso, el gasto es igual y se debe comparar a las personas (de tipo nat). Como son dos comparaciones de naturales, asumidas $\mathcal{O}(1)$, esto tiene complejidad $\mathcal{O}(1)$.

$>_{\alpha}$ (**in** *puetoid1* : *puetoid*, **in** *puetoid2* : *puetoid*) $\rightarrow res$: bool

1: $res \leftarrow puetoid1 < puetoid2$

Complejidad: $\mathcal{O}(1)$

Justificacion: Es una comparacion de naturales, asumida $\mathcal{O}(1)$

3. Módulo Puesto de Comida

Interfaz

Usa: `diccLog(a, b)`, `item`, `cant`, `nat`, `puesto`, `dinero`, `persona`, `tupla`, `precio`, `bool`

Se explica con: `PUESTODECOMIDA`

géneros: `puesto`

Operaciones:

CREARPUESTO(`in p: dicc(item, nat)`, `in s: dicc(item, nat)`, `in d: dicc(item, dicc(cant, nat))`) $\rightarrow res : puesto$
Pre $\equiv \{claves(p) = claves(s) \wedge claves(d) \subseteq claves(p)\}$
Post $\equiv \{res =_{obs} crearPuesto(p, s, d)\}$
Complejidad: $\mathcal{O}(I^2 + \sum_{i' \in i} (\#DescuentosDelItem_{i'}^2 + MaximaCantConDescuento^2) + I * \log(I))$

Descripción: Inicializa un nuevo puesto, con un menú con precios, stock de productos y set de descuentos asociados a una cantidad mínima comprada de un productos predefinidos, ya que son todos pasados como parámetros de entrada.

Aliasing: Los parametros de entrada se pasan como referencia no modificable, y se devuelve una referencia modificable a `res`.

STOCK(`in p: puesto`, `in i: item`) $\rightarrow res : cant$

Pre $\equiv \{i \in menu(p)\}$
Post $\equiv \{res =_{obs} stock(p, i)\}$
Complejidad: $\mathcal{O}(\log(I))$

Descripción: Obtiene el stock de un ítem de un puesto. El ítem debe existir en el menú de dicho puesto.

Aliasing: Devuelve una referencia modificable al stock de un determinado puesto, este se ve afectado al hackear en Lollapatuza y al recibir una compra en Lollapatuza

DESCUENTO(`in p: puesto`, `in i: item`, `in c: cant`) $\rightarrow res : nat$

Pre $\equiv \{i \in menu(p)\}$
Post $\equiv \{res =_{obs} descuento(p, i, c)\}$
Complejidad: $\mathcal{O}(\log I)$

Descripción: Obtiene el descuento de un ítem dada la cantidad del mismo, si tal descuento existe. El descuento cumple que se aplica a una cantidad menor o igual que la cantidad dada, pero no hay otro descuento aplicado a una cantidad menor o igual a `cant` pero mayor al del resultado.

Aliasing: Los parámetros de entrada se pasan como referencia no modificable, y `res` es natural, por lo que se pasa por copia

GASTODE(`in p: puesto`, `in a: persona`) $\rightarrow res : dinero$

Pre $\equiv \{True\}$
Post $\equiv \{res =_{obs} precio(p, a)\}$
Complejidad: $\mathcal{O}(\log A)$

Descripción: Obtiene el gasto realizado por una persona en el puesto.

Aliasing: Los parámetros de entrada se pasan como referencia no modificable, y `res` es natural, por lo que se pasa por copia

Auxiliares

AGREGARVENTA(`in/out p: puesto`, `in a: persona`, `in i: item`, `in c: cantidad`) $\rightarrow res : tupla(itDiccLog, nat, nat)$

Pre $\equiv \{i \in \text{menu}(p) \wedge_L \text{haySuficiente}(p, i, c) \wedge p =_{\text{obs}} p_0\}$
Post $\equiv \{p =_{\text{obs}} \text{vender}(p, a, i, c) \wedge (\pi_1(\text{res}) \neq \text{nullptr}) \Rightarrow_L (\text{siguiente}(\pi_1(\text{res})) = p \wedge \pi_1(\text{siguiente}(\pi_2(\text{res}))) = i \wedge \pi_2(\text{siguiente}(\pi_2(\text{res}))) = c \wedge \pi_3(\text{res}) = \text{AplicarDescuento}(i^*c, \text{descuento}(p, i, c)))\}$
Complejidad: $\mathcal{O}(\log(A))$

Descripción: Vende la cantidad dada del producto que debe encontrarse en el menu del puesto dado, aplicando el descuento si tal existe y removiendo la cantidad dada del stock. En caso de ser una venta sin descuento, devuelve, en res iteradores hacia el puesto como primer elemento, el 2do elemento es el precio de la compra y el 3er elemento de la tupla es la cantidad comprada.

Aliasing: El puesto es pasado como referencia modificable, mientras que los restantes parámetros son pasados como referencias no modificables. Res consiste de un iterador que puede modificar al puesto (pero el iterador en si no es modificable), pero solo en caso de que la compra sea sin descuento. En caso de que la compra tenga un descuento, este iterador es nullptr, ya que solo queremos acceder de vuelta al puesto en caso de querer modificar una compra sin descuento

ELIMINARVENTASINDESCUENTO(in/out p : puesto, in a : persona, in i : item, in pre : precio)

Pre $\equiv \{\text{La persona dada tiene una compra en el puesto dado, en el cual se compra el item dado y el precio del item es el dado. La compra es una de las compras sin descuento de la persona(es decir, debe existir tal compra). El precio del item corresponde al precio de item del menu del puesto}\}$

Post $\equiv \{\text{El stock del item dado para el puesto dado es incrementado en 1, la venta a la que apuntaba el iterador inicial es modificada: se decrementa en uno la cantidad y el precio corresponde al precio de remover uno del item dado de la compra. El gasto en el puesto de la persona es reducido solamente por el precio del item dado. Ningun otro atributo del puesto es modificado.}\}$

Complejidad: $\mathcal{O}(\log(A) + \log(I))$

Descripción: Esta operación realiza el efecto del hackeo al nivel del puesto: agrega uno del item al stock, modifica a la persona referida en la compra reduciendo su gasto total en una unidad del item dado.

Aliasing: El puesto de comida es un parametro que se pasa por referencia modificable. El resto de los parametros de entrada son referencias no modificables.

APLICARDESCUENTO(in $descuento$: nat, in $nuevoPrecio$: precio) $\rightarrow res$: nat

Pre $\equiv \{\text{descuento} < 100\}$

Post $\equiv \{res =_{\text{obs}} \text{aplicarDescuento}(nuevoPrecio, descuento)\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve el precio resultante de aplicar un descuento dado a un precio dado

Aliasing: Los parámetros de entrada se pasan por copia y res se pasa por copia

Representación

Representación de PuestoDeComida:

puesto se representa con *estr*

donde *estr* es $\text{tupla}(\text{men: diccLog}(\text{item}, \text{tupla}(\text{precio}, \text{stock}, \text{descuentos: arreglo}[1..n] \text{ de nat}))$
 $, \text{ventasPorPersona: diccLog}(\text{persona}, \text{tupla}(\text{gastoTotal: nat}, \text{ventas: secuencia}(\text{tupla}(\text{item}, \text{cantidad}))))$
 $)$

donde *item*, *precio*, *stock*, *cantidad*, *descuento*, *persona*, *dinero* son NAT

Invariante de representación

- 1) Todos los descuentos de un item, para cualquier cantidad son menores estrictos a 100.
- 2) La suma de todas las ventas hacia una persona, al aplicar los descuentos correspondientes, debería ser mayor o igual al gasto total de dicha persona. (Ya que hackeo modifica el gasto total, pero no las ventas)
- 3) Para toda persona que tiene al menos una compra registrada en ventas, todos los items que forman parte de las compras son items que estan en el menu del puesto.

$\text{Rep} : \text{estr} \rightarrow \text{bool}$

$\text{Rep}(e) \equiv \text{true} \iff$
 $\text{DescuentosValidos}(e) \wedge$
 $\text{ComprasValidas}(e)$

$\text{Abs} : \text{estr } e \rightarrow \text{PuestoDeComida} \quad \{\text{Rep}(e)\}$

$\text{Abs}(e) \equiv (\forall p : \text{PuestoDeComida})(p =_{\text{obs}} e /$
 $\text{menu}(p) = \text{claves}(e.\text{menu}) \wedge$
 $(\forall i : \text{item})(\text{precio}(p, i) = \text{obtener}(e.\text{menu}, i).\text{precio})$
 $(\forall i : \text{item})(\text{stock}(p, i) = \text{obtener}(e.\text{menu}, i).\text{stock})$
 $(\forall i : \text{item}, c : \text{cant})(\text{descuento}(p, i, c) =$
 $\text{if } \text{tam}(e.\text{descuentos}) = 0 \text{ then}$
 0
 else
 $\text{if } c > \text{tam}(e.\text{descuentos}) \text{ then}$
 $e.\text{descuentos}[\text{tam}(e.\text{descuentos})]$
 else
 $e.\text{descuentos}[c]$
 end if
 end if
 $\wedge (\forall per : \text{persona})(\text{ventas}(p, per) =$
 $\text{secuenciaAMulticonjunto}(\text{obtener}(e.\text{ventasPorPersona}, per).\text{ventas}))$

DescuentosValidos : estr \longrightarrow bool

DescuentosValidos($e : \text{estr}$) {
 $(\forall i : \text{item})(\text{def?}(i, e.\text{menu}) \Rightarrow_L ((\forall c : \text{cantidad})(0 < c \leq \text{tam}(\text{descuentos})) \Rightarrow_L$
 $\text{descuentos}[c] < 100))$ }
}

ComprasValidas : estr \longrightarrow bool

ComprasValidas($e : \text{estr}$) {
 $(\forall p : \text{persona})(p \in \# \text{claves}(e.\text{ventasPorPersona}) \Rightarrow_L$
 $((\forall t : \text{tupla}(\text{item}, \text{cantidad}))(\text{está?}(t, \text{obtener}(p, e.\text{ventasPorPersona}).\text{ventas}) \Rightarrow_L$
 $\pi_1(t) \in \text{claves}(e.\text{menu})) \wedge$
 $\text{obtener}(p, e.\text{ventasPorPersona}).\text{gastoTotal} \leq$
 $\text{sumaDeAplicarDescuentoATodo}(e.\text{menu}, \text{obtener}(p, e.\text{ventasPorPersona}).\text{ventas}))$
}

$\text{sumaDeAplicarDescuentoATodo} : \text{menu} \times \text{secu}(\text{tupla}(\text{item} \times \text{cantidad})) \longrightarrow \text{nat}$
{todo item de toda compra esta en el menu}

sumaDeAplicarDescuentoATodo(m, s) {
if vacia?(s) **then** 0
else AplicarDescuento(significado($\pi_1(\text{prim}(s)), e.\text{menu}$)/ $\pi_2(\text{prim}(s))$,
descuento($m, \pi_1(\text{prim}(s)), \pi_2(\text{prim}(s))$)) + sumaDeAplicarDescuentoATodo($m, \text{fin}(s)$)
endif
}

$\text{descuento} : \text{menu} \times \text{item} \times \text{cantidad} \longrightarrow \text{nat}$
{todo item de toda compra esta en el menu}

descuento(m, i, c) {
if $c = 0$ **then** 0
else if $m > \text{tam}(m.\text{descuentos})$ **then** descuento($m, i, c-1$)
else $m.\text{descuentos}[c]$
endif
}

Algoritmos

```

iCREARPUESTO(in  $p$ : dicc(item, nat), in  $s$ : dicc(item, nat), in  $d$ : dicc(item,
dicc(cant, nat)))  $\rightarrow res$ : estr
1:  $res \leftarrow \langle \text{Vacío}(), \text{Vacío}() \rangle$ 
2:  $it \leftarrow \text{crearIt}(p)$ 
3: while haySiguiente( $it$ ) do
4:    $precio \leftarrow \text{significado}(p, \text{siguiente}(it))$ 
5:    $stock \leftarrow \text{significado}(s, \text{siguiente}(it))$ 
6:    $descuentos \leftarrow \text{Vacía}()$ 
7:   if definido?( $d, \text{siguiente}(it)$ ) then
8:      $\text{DescuentosaCopiar} \leftarrow \text{significado}(d, \text{siguiente}(it))$ 
9:      $itDescuentos \leftarrow \text{CrearIt}(\text{DescuentosACopiar})$ 
10:    while haySiguiente( $itDescuentos$ ) do
11:       $sign \leftarrow \text{significado}(\text{DescuentosACopiar}, \text{siguiente}(it))$ 
12:      while longitud( $descuentos$ )  $\leq$  siguiente( $it$ ) do AgregarAtras( $descuentos, 0$ )
13:      end while
14:       $descuentos[\text{siguiente}(it)] = sign$ 
15:      Avanzar( $itDescuentos$ )
16:    end while
17:     $i \leftarrow 1$ 
18:     $descuentosMenores \leftarrow 0$ 
19:    while  $i \leq \text{longitud}(descuentos)$  do
20:      if  $descuentos[i] \neq 0$  then
21:         $descuentosMenores \leftarrow descuentos[i]$ 
22:      end if
23:       $descuentos[i] = descuentosMenores$ 
24:       $i \leftarrow i+1$ 
25:    end while
26:  end if
27:  definirRapido( $res.menu, \text{siguiente}(it), \langle precio, stock, descuentos \rangle$ )
28:  Avanzar( $it$ )
29: end while
30: return  $res$ 
Complejidad:  $\mathcal{O}(I^2 + \sum_{i \in i} (\#DescuentosDelItem_i^2 + \text{MaximaCantConDescuento}^2) + I \cdot \log(I)) = 0$ 

```

Justificacion: Primero que nada, crear diccionarios Vacios, crear iteradores a diccionarios y avanzar dichos iteradores es $\mathcal{O}(1)$. Lo mismo asumimos de copiar y comparar numeros naturales. En el peor caso, dado una cantidad de items i , todo item tiene un descuento. Como debemos obtener tanto el precio, stock y, a la vez ver si tiene definidos los descuentos de todo item y obtenerlos, debo sumar las complejidades de significado de estos 3, y el de definido? de descuentos. Asumimos que los diccionarios de entrada están diseñados como ABB, por lo que buscar el significado o si esta o no definido es, en peor caso, $\mathcal{O}(I)$, por lo que buscar significado de todos los elementos de un diccionario es $\mathcal{O}(\text{maximoDescuento}^2)$. Finalmente, definimos los descuentos para todo item. Como trabajamos con vectores, el peor caso posible es el cual en los descuentos se incorporan en orden ascendente, y se debe reservar memoria y copiar el vector para todo descuento nuevo, en cuyo caso se suma el pusheo de todos los indices hasta el nuevo descuento maximo. En peor caso esto se hace en cada agregado, por lo que acoto por el descuento de la cantidad máxima al cuadrado. Finalmente, para definir cada clave del nuevo diccionario de items, podemos usar definirRapido, ya

que nos aseguramos de que no hay igualdad de claves por que los diccionarios son parámetros de entrada, así evitamos comparar la igualdad. La insercion rápida en un DiccLog es $\mathcal{O}(\log(I))$, (i es el tamaño actual del diccionario) ya que evitamos ver la igualdad y solo buscamos la ubicación del nuevo elemento. Nos queda entonces una sumatoria de $\log(I)$, con i desde 1 hasta la cantidad de claves, que sabemos que se puede acotar por la cantidad de items, por el logaritmo de la cantidad de items (aproximacion de Stirling).

Resumen: Primero crea el esqueleto de la estructura: dos diccionarios vacíos. Itera por todos los items que son claves de los diccionarios (las claves de diccionarios de precios y stock son iguales) y, para cada item, primero obtiene su stock inicial y su precio. Para definir los descuentos, usamos un arreglo: si el item no tiene descuentos (es decir, no está definido en el diccionario), el arreglo será vacío. Si tiene al menos un descuento definido, va por todos los descuentos y agrega elementos al arreglo hasta que tenga la longitud del mayor descuento. Para las cantidades que tienen un descuento para una cantidad estrictamente igual, define el descuento, sino, define el descuento como 0. Luego hace una segunda pasada por el arreglo del descuento, y va guardando en cada posicion el descuento para la mayor cantidad menor (o 0 si la cantidad es menor al primer descuento). Finalmente, define en el menu del puesto como los precios, el stock y los descuentos de cada item. El otro elemento del puesto, ventasPorPersona, queda vacío ya que no hay persona alguna en los parámetros de entrada.

iSTOCK(in e : **estr**, in i : **item**) $\rightarrow res$: **cant**

1: $res \leftarrow \text{significado}(e.\text{menu}, i).\text{stock}$
2: return res

Complejidad: $\mathcal{O}(\log(I))$

Justificacion: Buscar el significado de una clave en $e.\text{menu}$ tiene complejidad $\mathcal{O}(\log(I))$ siendo i la cantidad de items definidos, pues este es un diccionario logaritmico.

iDESCUENTO(in p : **estr**, in i : **item**, in c : **cant**) $\rightarrow res$: **nat**

1: $\text{diccItem} \leftarrow \text{significado}(e.\text{menu}, i)$
2: **if** $\text{tam}(\text{diccItem}.\text{descuentos}) = 0$ **then**
3: $res \leftarrow 0$
4: **else**
5: **if** $c > \text{tam}(\text{diccItem}.\text{descuentos})$ **then**
6: $res \leftarrow \text{diccItem}.\text{descuentos}[\text{tam}(\text{diccItem}.\text{descuentos})]$
7: **else**
8: $res \leftarrow \text{diccItem}.\text{descuentos}[c]$
9: **end if**
10: **end if**
11: return res
12: Complejidad: $\mathcal{O}(\log I)$

Justificacion: El algoritmo consiste únicamente de comparaciones de naturales, acceso al tamaño de un vector y acceso a posiciones indexadas del vector, todas operaciones en $\mathcal{O}(1)$, ademas de tener que buscar el item dentro del puesto, la cual, al estar diseñado con un DiccLog, es $\mathcal{O}(\log(I))$, siendo esta la complejidad del algoritmo.

Resumen: Hay 3 casos posibles: la cantidad es menor al primer descuento de un item, por lo que el descuento es 0, o la cantidad es mayor al ultimo descuento del item, por lo que se devuelve este. Si la cantidad está en el arreglo, devuelve el descuento de la posición "cantidad" del arreglo, ya que, como definimos en crear puesto, contiene el descuento correspondiente.

iGASTODE(in/out e : **estr**, in a : **persona**) $\rightarrow res$: **dinero**

```
1: if  $\neg$ definido?(e.ventasPorPersona,a) then  
2:    $res \leftarrow 0$   
3: else  
4:    $compras \leftarrow \text{significado}(e.ventasPorPersona, a)$   
5:    $res \leftarrow compras.gastoTotal$   
6: end if  
7: return  $res$ 
```

Complejidad: $\mathcal{O}(\log(A))$

Justificación: Primero se busca la persona de todas las personas que alguna vez realizaron una compra, lo cual es $\mathcal{O}(\log A)$, y luego se accede a un elemento de la tupla ($\mathcal{O}(1)$) Por ende, es $\mathcal{O}(\log(A))$. Si la persona nunca realizo una compra en el puesto, igual debe ver si está definida $\mathcal{O}(\log A)$ y devuelve 0.

Auxiliares

iAGREGARVENTA(in/out e : estr, in a : persona, in i : item, in c : cantidad) $\rightarrow res$:
tupla(itDiccLog,nat,nat)

```
1: producto  $\leftarrow$  significado( $e.menu, i$ )
2: producto.stock  $\leftarrow$  producto.stock -  $c$ 
3: descuento  $\leftarrow$  Descuento( $p, i, c$ )
4: nuevoPrecio  $\leftarrow$  producto.precio* $c$ 
5:  $res \leftarrow \langle nullptr, nullptr, 0 \rangle$ 
6: nuevoPrecio  $\leftarrow$  AplicarDescuento(descuento,nuevoPrecio)
7: if  $\neg(\text{definido?}(e.ventasPorPersona,a))$  then
8:   definir( $e.ventasPorPersona,a,\langle 0, Vacia() \rangle$ )
9: end if
10: individuo  $\leftarrow$  significado( $e.ventasPorPersona,a$ )
11: individuo.gastoTotal  $\leftarrow$  individuo.gastoTotal + nuevoPrecio
12: if descuento = 0 then
13:    $\pi_1(res)^* \leftarrow e$ 
14: end if
15:  $(\pi_2(res)) \leftarrow$  nuevoPrecio
16:  $(\pi_3(res)) \leftarrow c$ 
17: return  $res$ ;
```

Complejidad: $\mathcal{O}(\log(I) + \log(A))$

Justificacion: Primero se busca el item entre todos los items del menú del puesto, lo cual es $\mathcal{O}(\log I)$. Se accede al descuento que corresponde a dicho item en la cantidad dada ($\mathcal{O}(1)$) y luego se debe acceder a la persona que realiza la compra en ventasPorPersona(o definirla si no existe), pero en peor caso esto es $\mathcal{O}(2*\log(A)) = (\mathcal{O}(\log(A)))$, y, ademas llama a aplicarDescuento, el cual asumimos que es $\mathcal{O}(1)$. Como el resto del algoritmo consiste en asignar variables por referencia y hacer comparaciones de naturales, aseguramos que este algoritmo es $\mathcal{O}(\log(A) + \log(I))$

iELIMINARVENTASINDESCUENTO(in/out e : *estr*, in a : *persona*, in i : *item*, in p : *precioItem*)

- 1: producto \leftarrow significado(e .menu, i)
- 2: producto.stock \leftarrow producto.stock+1
- 3: per \leftarrow significado(e .ventasPorPersona, a)
- 4: per.gastoTotal \leftarrow per.gastoTotal - precioItem

Complejidad: $\mathcal{O}(\log(I) + \log(A))$

Justificacion: Primero se busca el item en todos los items del menú del puesto, lo cual es $\mathcal{O}(\log I)$ para poder incrementar su stock. Luego se debe acceder a la persona que realiza la compra en ventasPorPersona para reducir el gasto total en el consumo de un item de la compra, el cual en peor caso es $\mathcal{O}(\log(A))$. El resto de las operaciones es $\mathcal{O}(1)$, por lo que la complejidad es $\mathcal{O}(\log(I) + \log(A))$

Resumen: Esta funcion es bastante simple, lo que vale la pena mencionar es que esta rompe la consistencia entre lollapatuza y el puesto, y no cumple plenamente con el TAD puesto, ya que, si bien modifica el gasto de la persona en el puesto, no modifica de igual forma las compras de la persona en este. Esta fue una decisión tomada en la reentrega del TP, ya que tomar esto en cuenta complejizaba demasiado la estructura de lollapatuza y sus algoritmos.

iAplicarDescuento(in *descuento*: *nat*, in *nuevoPrecio*: *precio*) $\rightarrow res$: *nat*

- 1: res \leftarrow (nuevoPrecio - ((nuevoPrecio \times descuento) \div 100))

Complejidad: $\mathcal{O}(1)$

Justificacion: Se asume que multiplicación y división de punto flotante son $\mathcal{O}(1)$.

Se asume que la division es flotante y se realiza en $\mathcal{O}(1)$, asi siempre aplicar un 0 % de descuento mantiene el valor del producto y se cumple la cota de complejidad de vender.

4. Módulo ColaDePrioridad(α)

Se explica con: COLA DE PRIORIDAD

géneros: ColaPrioridad

Interfaz

Usa: bool, α

Parametros formales

generos α

COPIAR(**in** $a : \alpha$) $\rightarrow res : \alpha$

Pre $\equiv \{\text{True}\}$

Post $\equiv \{res =_o bs \text{ Copiar}(a)\}$

Complejidad: $O(\text{copy}(a))$

Descripción: Se devuelve una copia del elemento

Aliasing: Crea una copia del elemento

$>_\alpha$ (**in** $a1 : \alpha$, **in** $a2 : \alpha$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{True}\}$

Post $\equiv \{res =_o bs \ a1 >_\alpha a2\}$

Complejidad: $O((>_\alpha)(a1, a2))$

Descripción: Los parámetros se pasan por referencia no modificable.

Aliasing: Devuelve true si y solo si, dada una relacion de orden, el primer parámetro es mayor al segundo

Operaciones:

VACIA?(**in** $c : \text{colaPrioridad}(\alpha)$) $\rightarrow res : \text{bool}$

Pre $\equiv \{\text{True}\}$

Post $\equiv \{res =_{obs} \text{Vacía?}(c)\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve true si y solo si la cola de prioridad dada no tiene elementos encolados

Aliasing: Los parámetros de entrada son pasados por referencia no modificable

PROXIMO(**in** $c : \text{colaPrioridad}(\alpha)$) $\rightarrow res : \alpha$

Pre $\equiv \{\neg \text{Vacía?}(c)\}$

Post $\equiv \{res =_{obs} \text{proximo}(c)\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Devuelve el siguiente elemento a desencolar de una cola de prioridad no vacia

Aliasing: Res es modificable si y solo si c lo es

DESENCOLAR(**in/out** $c : \text{colaPrioridad}(\alpha)$)

Pre $\equiv \{\neg \text{Vacía?}(c)\}$

Post $\equiv \{res =_{obs} \text{desencolar}(c)\}$

Complejidad: $\mathcal{O}(\log(\text{tamaño}(c)))$

Descripción: Desencola el elemento de más prioridad de la cola, asumiendo que se da una cola no vacia

Aliasing: c entra y sale como referencia modificable

VACIA() $\rightarrow res : \text{colaPrioridad}(\alpha)$

Pre $\equiv \{\text{True}\}$

Post $\equiv \{res =_{obs} \text{Vacía}()\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Crea una cola de prioridad sin elementos

Aliasing: Se devuelve una referencia modificable a res

ENCOLAR(**in/out** $c : \text{colaPrioridad}(\alpha)$, **in** $a : \alpha \rightarrow \text{res} : \text{puntero}(\text{colaPrioridad}(\alpha))$)

Pre $\equiv \{c =_o bs\ c_0\}$

Post $\equiv \{c =_{obs} \text{encolar}(c, a) \wedge *res.raiz =_o bs\ a\}$

Complejidad: $\mathcal{O}(\log(\text{tamaño}(c)))$

Descripción: Encola el elemento a según su prioridad con respecto al resto de los elementos de la cola, de haberlos. Caso contrario, se inserta como elemento de más prioridad

Aliasing: c y a se pasan como referencias, donde c es modificable, a no lo es, y res se pasa como referencia modificable

Auxiliares

SIFTUP(**in/out** $c : \text{colaPrioridad}(\alpha)$, **in/out** $a : \text{ArbolConPadre}(\alpha) \rightarrow \text{res} : \text{itColaPrioridad}$

Pre $\equiv \{a \text{ es un elemento de la cola de prioridad, y todos sus atributos coinciden con los del nodo de } a \text{ en } c\}$

Post $\equiv \{\text{La cola es una permutacion de la pasada por entrada, en la que se cumple el invariante de representacion de un heap}\}$

Complejidad: $\mathcal{O}(\prod n \in \text{caminoHastaRaizDesdeA}(>_\alpha(n.raiz, n.padre.raiz)))$

Descripción: Swapea la posición de A con la de su padre, mientras A tenga padre y A tenga mayor prioridad a la de este

Aliasing: Los parámetros de entrada se pasan por referencia modificable y devuelve una referencia modificable a la cola de prioridad

SIFTDOWN(**in/out** $c : \text{colaPrioridad}(\alpha)$, **in/out** $a : \text{arbolConPadre}(\alpha)$)

Pre $\equiv \{a \text{ es un elemento de la cola de prioridad, y todos sus atributos coinciden con los del nodo de } a \text{ en } c\}$

Post $\equiv \{\text{La cola es una permutacion de la pasada por entrada, en la que se cumple el invariante de representacion de un heap}\}$

Complejidad: $\mathcal{O}(\prod n \in \text{maxCaminoHastaPosicionDesdeA}(>_\alpha(n.raiz, *(n.der).raiz) + >_\alpha(n.raiz, *(n.izq).raiz) + >_\alpha(*(n.izq).raiz, *(n.der).raiz)))$

Descripción: Swapea la posición de a con la del mayor de sus hijos, mientras A tenga hijos y alguno de ellos tenga mayor prioridad

Aliasing: Los parámetros de entrada se pasan por referencia modificable, y devuelve una referencia modificable a una posición de la cola de prioridad

SWAP(**in/out** $c : \text{colaPrioridad}(\alpha)$, **in/out** $\text{Mayor} : \text{arbolConPadre}(\alpha)$, **in/out** $\text{Menor} : \text{arbolConPadre}(\alpha) \rightarrow \text{res} : \text{tupla}(\text{arbolConPadre}(\alpha), \text{arbolConPadre}(\alpha))$)

Pre $\equiv \{\text{Mayor y Menor son elementos de la cola de prioridad, que se asocian de forma que Mayor es padre de Menor y uno de los hijos de Mayor es menor}\}$

Post $\equiv \{\text{La cola es una permutacion de la pasada por entrada, en la que se cumple que, luego del swap, Menor es padre de Mayor, Mayor es alguno de los hijos de menor, los hijos de menor son ahora los de mayor y viceversa. Ningun otro nodo de la cola de prioridad se ve afectado. Res es igual a la tupla (NuevoMayor, NuevoMenor), que revierte los ordenes de los parametros de la entrada}\}$

Complejidad: $\mathcal{O}(1)$

Descripción: Swapea la posición de a con la del mayor de sus hijos, mientras A tenga hijos y alguno de ellos tenga mayor prioridad

Aliasing: Los parámetros de entrada se pasan por referencia modificable, y res es una referencia modificable

Nota: el punto de este algoritmo es que los punteros de la cola de prioridad mantengan la referencia al nodo con el valor con los que fueron inicializados. Puede estar mal diseñado el algoritmo, pero el objetivo del algoritmo es ese. El unico proposito de este algoritmo es para que vender y hackeo mantengan la consistencia en los punteros a los nodos de el que mas gasto, asi, una vez que se aumenta o disminuye la prioridad de un

individuo, el puntero al que apunta todo individuo en el puesto o cola de prioridad siga apuntando al mismo.

Representación

Representación de ColaPrioridad(α):

colaPrioridad(α) se representa con estr

donde **estr** es **tupla(másPrioridad: puntero(ArbolConPadre(α)) , tamaño: nat)**

donde ArbolConPadre(α) es tupla(
padre : puntero(ArbolConPadre(α),
izq : puntero(ArbolConPadre(α),
der : puntero(ArbolConPadre(α))

Invariante de Representacion

Dado que el arbol no es null:

- 1) El elemento de mayor prioridad no tiene padre
- 2) Tanto el hijo derecho como izquierdo de un nodo tienen como padre a dicho nodo. El hijo derecho, si no es nullptr, no puede apuntar a lo mismo que el hijo izquierdo y viceversa
- 3) La raiz del arbol tiene mayor prioridad, dada la relacion de orden, que sus hijos derecho e izquierdo, si tales existen
- 4) El arbol está balanceado
- 5) 2), 3) y 5) se cumplen recursivamente para los subarboles derecho e izquierdo
- 6) El arbol es izquierdista, es decir, llena los niveles de izquierda a derecha
- 7) El tamaño coincide con la cantidad de nodos del arbol
- 8) No se producen repeticiones de punteros(excepto de padre) en el arbol, es decir, cada puntero a hijo derecho e izquierdo de un nodo no se repite

Rep : estr \longrightarrow bool

Rep(e) \equiv true \iff
 másPrioridad = nullptr \vee_L (**RaizSinPadre**(e) \wedge
PadreDeHijos(*($e.raiz$)) \wedge
balanceado(*($e.raiz$)) \wedge
PadreMayorPrioridad(*($e.raiz$)) \wedge_L **HijosCumplen**(*($e.raiz$)) \wedge
izquierdista(*($e.raiz$)) \wedge **tamañoCoincide**($e.raiz, e.tamaño$) \wedge
sinPunterosRepetidos($e.raiz, e.tamaño$)

Abs : estr $e \longrightarrow$ ColaPrioridad(α) {Rep(e)}

Abs(e) \equiv ($\forall c : \text{colaPrioridad}(\alpha)$)($c =_{obs} e /$
 Vacía?(c) = ($e.masPrioridad = \text{nullptr}$) \wedge
 \neg Vacía?(c) \Rightarrow_L
 (proximo(c) = *($e.masPrioridad$).raiz \wedge
 Desencolar(c) = $e.sinMaximo$)

No especificamos sinMaximo, pero damos a entender que elimina el elemento de mayor prioridad de la estructura, de manera que el 2do elemento de mayor prioridad (si tal existe) ahora sea el de mayor prioridad, manteniendo el invariante de representacion. Si el elemento que borramos era el único, $e.masPrioridad$ será un nullptr

RaizSinPadre : estr \longrightarrow bool

RaizSinPadre($e : \text{estr}$) $\rangle \rangle$ {
 *($e.masPrioridad$).padre = nullptr; }

PadreDeHijos : ArbolConPadre(α) \longrightarrow bool

PadreDeHijos($a : \text{ArbolConPadre}(\alpha)$) {
 ($\forall (aIzq, aDer : \text{arbolConPadre}(\alpha))$)($(aIzq.padre \neq \text{nullptr} \wedge_L *(aIzq.padre) = a \iff$
 $a.izq \neq \text{nullptr} \wedge_L *(a.izq) = aIzq) \wedge (aDer.padre \neq \text{nullptr} \wedge_L *(aDer.padre) = a \iff$
 $a.der \neq \text{nullptr} \wedge_L *(a.der) = aDer)$ }

balanceado : ArbolConPadre(α) \longrightarrow bool

balanceado($a : \text{ArbolConPadre}(\alpha)$) { altura(*(a.izq)) == altura(*(a.der)) \vee altura(*(a.izq)) == altura(*(a.der))+1 \vee altura(*(a.izq)) == altura(*(a.der))-1 \wedge ($a.der \neq \text{nullptr} \Rightarrow_L \text{balanceado}(*(a.der)) \wedge (a.izq \neq \text{nullptr} \Rightarrow_L \text{balanceado}(*(a.izq)))$) }

altura : ArbolConPadre(α) \longrightarrow nat

altura($a : \text{ArbolConPadre}(\alpha)$) {
if $a = \text{NULL}$ **then** 0
else (1+maxaltura(*(a.izq)), altura(*(a.der)))
endif
 }

PadreMayorPrioridad : ArbolConPadre(α) \longrightarrow bool

PadreMayorPrioridad($a : \text{ArbolConPadre}(\alpha)$) {
 $a.izq \neq \text{nullptr} \Rightarrow_L a.raiz >_\alpha *(a.izq).raiz \wedge a.der \neq \text{nullptr} \Rightarrow_L a.raiz >_\alpha *(a.der).raiz$ }

$\text{HijosCumplen} : \text{ArbolConPadre}(\alpha) \rightarrow \text{bool}$
HijosCumplen($a : \text{ArbolConPadre}(\alpha)$) {
 $a.\text{izq} \neq \text{nullptr} \Rightarrow {}_L (\text{PadreDeHijos}(*a.\text{izq}) \wedge \text{PadreMayorPrioridad}(*a.\text{izq}) \wedge$
 $\text{HijosCumplen}(*a.\text{izq}) \wedge a.\text{der} \neq \text{nullptr} \Rightarrow {}_L (\text{PadreDeHijos}(*a.\text{der}) \wedge \text{PadreMa-}$
 $\text{yorPrioridad}(*a.\text{der}) \wedge \text{HijosCumplen}(*a.\text{der}))$ }

 $\text{izquierdista} : \text{ArbolConPadre}(\alpha) \rightarrow \text{bool}$
izquierdista($a : \text{ArbolConPadre}(\alpha)$) {
 $a = \text{Null} \vee {}_L \text{Nivel}(*a.\text{izq}) \geq \text{Nivel}(*a.\text{der}) \wedge a.\text{izq} \neq \text{nullptr} \Rightarrow {}_L \text{izquierdis-}$
 $\text{ta}(*a.\text{izq}) \wedge a.\text{der} \neq \text{nullptr} \Rightarrow {}_L \text{izquierdista}(*a.\text{der})$ }

 $\text{tamañoCoincide} : \text{puntero}(\text{ArbolConPadre}(\alpha)) \times \text{nat} \rightarrow \text{bool}$
tamañoCoincide($a : \text{puntero}(\text{ArbolConPadre}(\alpha)), n:\text{nat}$) {
 $n = a.\text{tamaño}$ }

 $\text{tamaño} : \text{puntero}(\text{ArbolConPadre}(\alpha)) \rightarrow \text{nat}$
tamaño($a : \text{puntero}(\text{ArbolConPadre}(\alpha))$) {
if $a = \text{nullptr}$ **then** 0
else $1 + \text{tamaño}(*a.\text{izq}) + \text{tamaño}(*a.\text{der})$
endif }

 $\text{sinPunterosRepetidos} : \text{puntero}(\text{ArbolConPadre}(\alpha)) \times \text{nat} \rightarrow \text{bool}$
sinPunterosRepetidos($pa : \text{puntero}(\text{ArbolConPadre}(\alpha)), n:\text{nat}$) {
 $\# \text{PunterosDelArbol}(pa) = n$ }

 $\text{PunterosDelArbol} : \text{puntero}(\text{ArbolConPadre}(\alpha)) \times \text{nat} \rightarrow \text{bool}$
sinPunterosRepetidos($pa : \text{puntero}(\text{ArbolConPadre}(\alpha)), n:\text{nat}$) {
if $pa = \text{nullptr}$ **then** \emptyset
else $\text{Ag}(pa, \text{PunterosDelArbol}(*pa.\text{der}) \cup \text{PunterosDelArbol}(*pa.\text{izq}))$
endif
}

Algoritmos

iVacia?(in $e : \text{estr}$) $\rightarrow res : \text{bool}$

- 1: $res \leftarrow e.\text{masPrioridad} = \text{nullptr}$
- 2: return res

Complejidad: $\mathcal{O}(1)$

Justificación: Simplemente compara un puntero al que se tiene acceso directo con nullptr, comparación que se asume que tarda tiempo constante

iProximo(in $e : \text{estr}$) $\rightarrow res : \alpha$

- 1: $res \leftarrow *(e.\text{masPrioridad}).\text{raiz}$
- 2: return res

Complejidad: $\mathcal{O}(1)$

Justificación: Buscar la raíz del elemento con mayor prioridad, al cual, asumiendo que esta definido, se tiene acceso en $\mathcal{O}(1)$, ya que no necesito realizar una búsqueda para encontrarlo.

iDesencolar(in/out e : estr)

```
1: tam  $\leftarrow$  e.tamaño
2: if tam = 1 then
3:    $e.masPrioridad \leftarrow \text{nullptr}$ 
4: else
5:   actual  $\leftarrow$  *(e.masPrioridad)
6:   temp  $\leftarrow$  tamaño
7:   listaBinario  $\leftarrow$  Vacía()
8:   while temp > 0 do
9:     AgregarAdelante(listaBinario, temp mod 2)
10:    temp  $\leftarrow$  temp div 2
11:   end while
12:   it  $\leftarrow$  (Siguiente(CrearIt(listaBinario)))
13:   while haySiguiente(haySiguiente((it)) do
14:     if siguiente(it) = 1 then
15:       actual  $\leftarrow$  *(actual.der)
16:     else
17:       actual  $\leftarrow$  *(actual.izq)
18:     end if
19:   end while
20:   if siguiente(it) = 1 then
21:     actual  $\leftarrow$  *(actual.der)
22:     *(actual.padre).der  $\leftarrow \text{nullptr}$ 
23:   else
24:     actual  $\leftarrow$  *(actual.izq)
25:     *(actual.padre).izq  $\leftarrow \text{nullptr}$ 
26:   end if
27:   actual.padre  $\leftarrow \text{nullptr}$ 
28:   actual.izq  $\leftarrow$  *(e.masPrioridad).izq
29:   actual.der  $\leftarrow$  *(e.masPrioridad).der
30:   *(e.masPrioridad).izq  $\leftarrow \text{nullptr}$ 
31:   *(e.masPrioridad).der  $\leftarrow \text{nullptr}$ 
32:   if actual.der  $\neq \text{nullptr}$  then
33:     (*(actual.der).padre)  $\leftarrow$  actual
34:   end if
35:   if actual.izq  $\neq \text{nullptr}$  then
36:     (*(actual.izq).padre)  $\leftarrow$  actual
37:   end if
38:   *(e.masPrioridad)  $\leftarrow$  actual
39:   actual2  $\leftarrow$  *(e.masPrioridad)
40:   siftDown(e, actual2)
41: end if
42: e.tamaño  $\leftarrow$  e.tamaño-1
```

Complejidad: $\mathcal{O}(\log(e.tamaño))$

Justificación: Primero se hace una representación binaria del tamaño del heap, lo cual es $\mathcal{O}(\log(e.tamaño))$, asumiendo que div y mod son $\mathcal{O}(1)$. Luego se busca el último elemento del heap que, dado por su invariante de representación, se puede obtener mediante la representación binaria del tamaño del heap, y recorro las ramas hasta llegar a este. Como se sabe que camino lleva al último nodo, solo se recorren $\mathcal{O}(\log(e.tamaño))$ elementos. Se reemplaza el valor del elemento de mayor prioridad con el último que, como se pasan por referencia, es $\mathcal{O}(1)$. Finalmente, mientras la raíz del otrora último elemento sea menor al de alguno de los hijos de su posición actual, reemplaza la posición por la del mayor de sus hijos. En peor caso, se tiene que hacer reemplazos hasta una hoja del heap, por lo que se producen a lo sumo $\mathcal{O}(\log(e.tamaño))$ iteraciones. Por ende, al sumar las complejidades, la complejidad de este algoritmo es $\mathcal{O}(\log(e.tamaño))$

iVacía() $\rightarrow res : \text{estr}$

1: $res \leftarrow \langle \text{nullptr}, 0 \rangle$

Complejidad: $\mathcal{O}(1)$

Justificación: Se hace una asignación de un puntero nulo y un natural, juntos en una tupla. Como la copia de naturales es $\mathcal{O}(1)$ y asignar un puntero a nullptr se asume $\mathcal{O}(1)$, esto es $\mathcal{O}(1)$

iEncolar(in/out $e : \text{estr}$, in $a : \alpha$) $\rightarrow res : \text{itColaPrioridad}$

1: $tam \leftarrow e.tamaño$

2: **if** $tam = 0$ **then**

3: $*(e.masPrioridad) \leftarrow \langle a, \text{nullptr}, \text{nullptr}, \text{nullptr} \rangle$

4: **else**

5: $actual \leftarrow *(e.masPrioridad)$

6: $temp \leftarrow tamaño + 1$

7: $listaBinario \leftarrow Vacía()$

8: **while** $temp > 0$ **do**

9: $AgregarAdelante(listaBinario, temp \bmod 2)$

10: $temp \leftarrow temp \div 2$

11: **end while**

12: $it \leftarrow (\text{Siguiente}(\text{CrearIt}(listaBinario)))$

13: **while** $haySiguiente(haySiguiente(it))$ **do**

14: **if** $siguiente(it) = 1$ **then**

15: $actual \leftarrow *(actual.der)$

16: **else**

17: $actual \leftarrow *(actual.izq)$

18: **end if**

19: **end while**

20: **if** $siguiente(it) = 1$ **then**

21: $*(actual.der) \leftarrow \langle \text{puntero}(actual), \text{nullptr}, \text{nullptr}, a \rangle$

22: $actual \leftarrow actual.der^*$

23: **else**

24: $*(actual.izq) \leftarrow \langle \text{puntero}(actual), \text{nullptr}, \text{nullptr}, a \rangle$

25: $actual \leftarrow *(actual.der)$

26: **end if**

27: $actual \leftarrow \text{siftUp}(e, actual)$

28: **end if**

29: $e.tamaño \leftarrow e.tamaño + 1$

30: $res \leftarrow \text{puntero}(actual)$

31: **return** res

Complejidad: $\mathcal{O}(\log(e.tamaño))$

Justificación: Se asume que operaciones de enteros como división y módulo son $\mathcal{O}(1)$. Primero se crea una lista enlazada con la representación binaria del tamaño + 1, lo cual es $\mathcal{O}(\log(e.tamaño + 1))$, porque en estas listas agregar elementos al inicio es $\mathcal{O}(1)$. Luego se recorre, dada esta representación binaria, los nodos del heap hasta el padre del siguiente nodo, lo cual también es $\mathcal{O}(\log(e.tamaño))$. Finalmente, se hace un llamado a siftUp , el cual es $\mathcal{O}(\log(e.tamaño))$ en peor caso. Como las raíces, para tipos no primitivos, se pasan por referencia, no hay complejidad agregada por copiar elementos.

Auxiliares

iSiftUp(in/out e : **estr**, in/out a : **arbolConPadre**(α)) $\rightarrow res$: puntero(**arbolConPadre**(α))

act $\leftarrow a$

while act $\neq *(e.masPrioridad) \wedge act.raiz >_{\alpha} act.padre * .raiz$ **do**

act $\leftarrow \pi_1(\text{swap}(*(act.padre), act))$

end while

res $\leftarrow \text{puntero}(act)$

return res

Complejidad: $O(\prod_{n \in caminoHastaRaizDesdeA} (>_{\alpha} (n.raiz, n.padre.raiz)))$

Justificacion: Para toda iteracion del ciclo, se debe comparar el valor de la raiz del actual con el de su padre. En el caso mas frecuente de llamado de esta funcion, se parte de una hoja, en cuyo peor caso se llega a la raiz, ($\log(e.tamaño)$ iteraciones), y las comparaciones son $O(1)$, por lo que esto se resume a $O(\log(e.tamaño))$ en dichos casos

iSiftDown(in/out c : **ColaDePrioridad**(α), in/out a : **ArbolConPadre**(α))

act $\leftarrow a$

while act.der $\neq nullptr \wedge_L *(a.der).raiz >_{\alpha} a.raiz \vee a.izq \neq nullptr \wedge_L *(a.izq).raiz >_{\alpha} a.raiz$ **do**

act $\leftarrow a$

if act.izq = nullptr $\vee *(act.der).raiz >_{\alpha} *(act.izq).raiz$ **then**

act $\leftarrow \pi_2(\text{swap}(act, act.der * .raiz))$

else

if act.der = nullptr $\vee *(act.izq).raiz >_{\alpha} *(act.der).raiz$ **then**

act $\leftarrow \pi_2(\text{swap}(act, *(act.izq).raiz))$

end if

end if

end while

Complejidad: $O(\prod_{n \in maxCaminohastaPosicionDesdeA} (>_{\alpha} (n.raiz, *(n.der).raiz) + >_{\alpha} (n.raiz, *(n.izq).raiz) + >_{\alpha} (*(n.izq).raiz, *(n.der).raiz)))$

Justificacion: Para toda iteracion del ciclo, se debe comparar el valor de la raiz del actual con el de sus dos hijos, y los hijos entre si, si tales existen. En el caso mas frecuente de llamado de esta funcion, se parte de la raiz, en cuyo peor caso debemos reemplazar hasta la hoja ($\log(e.tamaño)$ iteraciones), y las comparaciones son $O(1)$, por lo que esto se resume a $O(\log(e.tamaño))$ en esos casos.

iSwap(in/out c : ColaDePrioridad(α), in/out $Mayor$: ArbolConPadre(α), in/out $Menor$: ArbolConPadre(α))

```
if Mayor.padre  $\neq$  nullptr then
    if (*(Mayor.padre).izq) = (Mayor) then
        (*(Mayor.padre).izq)  $\leftarrow$  Menor
    else
        (*(Mayor.padre).der)  $\leftarrow$  Menor
    end if
    *(Mayor.padre)  $\leftarrow$  Menor
    Menor.padre  $\leftarrow$  Mayor.padre
end if
if Menor.izq  $\neq$  nullptr then
    (*(Menor.izq).padre)  $\leftarrow$  Mayor
end if
if Menor.der  $\neq$  nullptr then
    (*(Menor.der).padre)  $\leftarrow$  Mayor
    if Mayor.izq  $\neq$  nullptr  $\wedge$  *(Mayor.izq) = Menor then
        temp  $\leftarrow$  Menor.izq
        Menor.izq  $\leftarrow$  Mayor.izq
        Mayor.izq  $\leftarrow$  temp
        Mayor.der  $\leftarrow$  Menor.der
        *(Menor.der)  $\leftarrow$  Mayor
        if Menor.izq  $\neq$  nullptr then
            (*(Menor.izq).padre) = Menor
        end if
    else
        if Mayor.der  $\neq$  nullptr  $\wedge$  *(Mayor.der) = Menor then
            temp  $\leftarrow$  Menor.der
            Menor.der  $\leftarrow$  Mayor.der
            Mayor.der  $\leftarrow$  temp
            Mayor.izq  $\leftarrow$  Menor.izq
            *(Menor.izq)  $\leftarrow$  Mayor
            if Menor.der  $\neq$  nullptr then
                (*(Menor.der).padre) = Menor
            end if
        end if
    end if
    res  $\leftarrow$  (Menor, Mayor)
```

Complejidad: $\mathcal{O}(1)$

Justificacion: Toda operacion del algoritmo trata sobre asignacion y comparación de punteros, lo cual se asume $\mathcal{O}(1)$.
